Scripting

You can either run the programs and gather the data in 2 minutes or 2 hours … you can pick

Oregon State
University

Mike Bailey

mjb@cs.oregonstate.edu

Oregon State
University
Computer Graphics

scripting.pptx

mjb – March 9, 2023

Why Are These Notes Here?

In this class, you are *required* to run your programs many times to observe the effect of different parameters on performance.

You could run those many versions one-at-a-time, but this could take hours. Or, you could write *scripts* that run all those parameter combinations in a couple of minutes.

So, if you are the kind of person who has loads of free time on their hands and has nothing else they want to do, feel free to use the slow one-at-a-time approach.

If you are not one of those, take a little time to learn how to do it via scripts.

Oregon State
University
Computer Graphics

mjb – March 9, 2023

Setting up Your Benchmarks to run from Scripts:
#1 -- the #define Approach

There are always advantages to not hardcoding constants into the middle of your program and, instead, setting them with a #define at the top where you can find that value and change it easily, like this:

```
#include <stdio.h>
#include <math.h>

#ifndef NUMT
#define NUMT            2
#endif

#ifndef NUMS
#define NUMS            32
#endif
```

Then, in the C or C++ program, all you have to do is use NUMT to, for example, set the number of threads, like this:

omp_set_num_threads(NUMT);

But, the use of the #ifndef/#endif construct has other advantages. It lets you either run this as a standalone program or run many occurrences of the program from a *script*.

Oregon State
University
Computer Graphics

mjb – March 9, 2023

Setting up Your Benchmarks to run from Scripts:
#1 -- the #define Approach

In our project assignments, you will run benchmarks, that is, you will try your application using several different combinations of parameters. Setting these combinations by hand inside your program one-at-a-time is a time-consuming pain. Your time is more valuable than that. Try doing it from a *script*.

In most C and C++ compilers, there is some mechanism to set a **#define** from outside the program. Most (all?) of them use the **-D** construct on the command line. So, we could create a file called *script.bash* that looks like this::

```
#!/bin/bash

#number of threads:
for  t  in 1 2 4 6 8
do
    echo NUMT = $t
    g++  -DNUMT=$t  prog.cpp  -o prog  -lm -fopenmp
    ./prog
done
```

Then, in the C or C++ program, all you have to do is use NUMT. For example:

omp_set_num_threads(NUMT);

This lets you automatically run your program 5 times with 1, 2, 4, 6, and 8 threads.

Oregon S
Univers To run this script, type: **bash script.bash**
Computer Graphics

mjb – March 9, 2023

Setting up Your Benchmarks to run from Scripts:
Method #1 -- the #define Approach

You can also test *multiple parameters* from the same script by nesting the loops. This one is done using **Bash Shell (*bash*)**:

```
#!/bin/bash

# number of threads:
for  t  in 1 2 4 6 8
do
     echo NUMT = $t
     # number of subdivisions:
     for s in 2  4  8  16  32  64  128  256  512  1024  2048  3072  4096
     do
          echo NUMS = $s
          g++  -DNUMS=$s  -DNUMT=$t   prog.cpp  -o prog  -lm  -fopenmp
          ./prog
     done
done
```

http://teespring.com

Oregon State
University
Computer Graphics

mjb – March 9, 2023

Or, in *C Shell (csh)*…

```
#!/bin/csh

# number of threads:
foreach  t  ( 1 2 4 6 8 )
     echo NUMT = $t
     # number of subdivisions:
     foreach s (  2  4  8  16  32  64  128  256  512  1024  2048  3072  4096 )
          echo NUMS = $s
          g++  -DNUMS=$s  -DNUMT=$t   prog.cpp  -o prog  -lm  -fopenmp
          ./prog
     end
end
```

To run this script, type: **csh script.csh**

Oregon State
University
Computer Graphics

mjb – March 9, 2023

Or, in Python…

```
import os

for  t  in [ 1, 2, 4, 6, 8 ]:
    print "NUMT = %d"  % t
    for s in [ 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 3072, 4096 ]:
        print "NUMS = %d" % s
        cmd = "g++  -DNUMS=%d  -DNUMT=%d   prog.cpp  -o prog  -lm  -fopenmp % ( s, t ) "
        os.system( cmd )
        cmd = "./prog"
        os.system( cmd )
```

To run this script, type: **python script.py**

Oregon State
University
Computer Graphics

mjb – March 9, 2023

Setting up Your Benchmarks to run from Scripts:
Method #2 -- the Command Line Arguments Approach

Instead of doing this:

```
#include <stdio.h>
#include <math.h>

#ifndef NUMT
#define NUMT                    8
#endif

#ifndef NUMS
#define NUMS                    32
#endif
```

Do this:

```
#include <stdio.h>
#include <math.h>

int NUMT =   8;

int NUMS = 32
```

Then, in the C or C++ program, all you have to do is use NUMT to set the number of threads like you did before, like this:

omp_set_num_threads(NUMT);

Oregon State
University
Computer Graphics

mjb – March 9, 2023

Now let's use argc and argv

When you write in C or C++, your *main* program, which is really a special function call, looks like this:

```
int main( int argc, char *argv[ ] )
{
        . . .
```

These arguments describe what was entered on the command line used to run the program.

The **argc** is the number of arguments (the arg **C**ount)

The **argv** is a list of argc character strings that were typed (the arg **V**ector).

The name of the program counts as the 0th argv (i.e., argv[0])

So, for example, when you type
    ls –l
in a shell, the *ls* program sees argc and argv filled like this:
    argc = 2
    argv[0] = "ls"
    argv[1] = "-l"

---

argc and argv

So, if NUMT and NUMS are global int variables:

```
        int NUMT =   2;
        int NUMS = 32;
```

and you want to set them to something else from the command line, like this:

./prog   1   64

Then, *inside your main program*, you would say this:
```
        if( argc >= 2 )
                NUMT = atoi(  argv[1]  );

        if( argc >= 3 )
                NUMS = atoi(  argv[2]  );
```

The if-statements guarantee that nothing bad happens if you forget to type values on the command line.

The *atoi* function converts a string into an integer ("ascii-to-integer").
If you ever need it, there is also an *atof* function for floating-point.

shared( ) in the *#pragma omp* Line

Also remember, if you use Method #2, then NUMS is a *variable*, and it needs to be declared as *shared* in the *#pragma omp* line:

#pragma omp parallel for default(none) shared(NUMS,xcs,ycs,rs,tn) reduction(+:numHits)

NUMT does not need to be declared in this way because it is not used in the for-loop that has the *#pragma omp* in front of it.

Oregon State
University
Computer Graphics

Setting up Your Benchmarks to run from Scripts:
Method #2 -- the Command Line Arguments Approach

In our project assignments, you will run benchmarks, that is, you will try your application using several different combinations of parameters. Setting these combinations by hand inside your program one-by-one is a time-consuming pain.
Your time is more valuable than that. Try doing it from a script.

```
#!/bin/bash

g++  prog.cpp  -o prog  -lm –fopenmp

#number of threads:
for  t  in 1 2 4 6 8
do
    echo NUMT = $t
    ./prog   $t
done
```
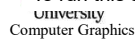
Then, in the C or C++ program, all you have to do is use NUMT. For example:

omp_set_num_threads(NUMT);

This lets you automatically run your program 5 times with 1, 2, 4, 6, and 8 threads.

To run this script, type: **bash script.bash**

University
Computer Graphics

Setting up Your Benchmarks to run from Scripts: #2 -- the Command Line Arguments Approach

You can also test multiple parameters from the same script by nesting the loops. This one is done using **Bash Shell (*bash*)**:

```
#!/bin/bash

g++  prog.cpp  -o prog  -lm  -fopenmp

# number of threads:
for  t  in 1 2 4 6 8
do
    echo NUMT = $t
    # number of subdivisions:
    for s in 2  4  8  16  32  64  128  256  512  1024  2048  3072  4096
    do
        echo NUMS = $s
        ./prog  $t  $s
    done
done
```

To run this script, type: **bash script.bash**

Oregon State
University
Computer Graphics

mjb – March 9, 2023

Or, in *csh (C Shell)* …

```
#!/bin/csh

g++  prog.cpp  -o prog  -lm  -fopenmp

# number of threads:
foreach  t  ( 1 2 4 6 8 )
    echo NUMT = $t
    # number of subdivisions:
    foreach s (  2  4  8  16  32  64  128  256  512  1024  2048  3072  4096 )
        echo NUMS = $s
        ./prog  $t  $s
    end
end
```

To run this script, type: **csh script.csh**

Oregon State
University
Computer Graphics

mjb – March 9, 2023

Or, in Python…

```
import os

cmd = "g++  prog.cpp  -o prog  -lm  -fopenmp"
os.system( cmd )

for  t  in [ 1, 2, 4, 6, 8 ]:
    print "NUMT = %d"  %  t
    for s in [ 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 3072, 4096 ]:
        print "NUMS = %d" % s
         cmd = "./prog  %d   %d"  %  ( s, t )
        os.system( cmd )
```

To run this script, type: **python script.py**

Oregon State
University
Computer Graphics

Do Not Put These Loops in the .cpp Program!

I know what you're thinking.

You're thinking:
"Those scripts are different, and I've never done them before, and I don't want to take the 5 minutes to learn them. So, I'll just build the iterations through all the parameters into for-loops in the program."

Don't!

I see evidence that the first time OpenMP does anything, it also does some one-time setups. This will mess up your timing because your first test will seem slower than it should be and the others will seem artificially faster by comparison.

I recommend you run the program *separately* for each combination of parameters. (The script code in the previous pages shows that.)

Oregon State
University
Computer Graphics

Diverting to a File

We all have a tendency to want to write our performance results out using *printf* (or *cout*) so that we can see them on the screen. That's fine. But, then we want to get those results into a file. You could mess with file I/O, or you could use a *divert* on the command line.

If you are currently running your program like this:

./proj01

and it prints to the standard output screen via *printf* or *cout*, then running it like this:

./proj01 > output.csv

will write your results into the file *output.csv*

(If you do it a second time, you will probably have to remove the previous *output.csv* first.)

You can also divert the *entire* output (standard out and standard error) of a looping ***script***:

bash script.bash >& output

Oregon State
University
Computer Graphics

mjb – March 9, 2023

Importing into Excel – csv Files

csv stands for **comma-separated values**. It is a file format where you write your numbers out as text with commas between them. The great part is that Excel recognizes csv files and will read them in automatically.

Say you are using a printf that looks like this:

```
printf( "%2d threads ; %8d trials ; probability = %6.2f%% ; megatrials/sec = %6.2lf\n",
        NUMT, NUMTRIALS, 100.*currentProb, maxPerformance);
```

You probably did this because it looks really nice on your screen as you use this output to debug your program. But, now you want to change it to get the numbers into Excel quickly and painlessly. Comment out the old way and change it to this:

```
//printf( "%2d threads ; %8d trials ; probability = %6.2f%% ; megatrials/sec = %6.2lf\n",
        //NUMT, NUMTRIALS, 100.*currentProb, maxPerformance);

 printf( "%2d, %8d, %6.2lf\n", NUMT, NUMTRIALS, maxPerformance );
```

This will now be printing just what you need in CSV format. You can divert it like this:

./proj01 > OUT.csv

or

bash script.bash > OUT.csv

Which would then let you read the OUT.csv file right into Excel.

mjb – March 9, 2023

Importing into Excel – csv Files

```
flip3 154% cat OUT.csv
1, 1, 1.44
1, 10, 3.99
1, 100, 8.07
1, 1000, 9.33
1, 10000, 23.40
1, 100000, 25.13
1, 500000, 25.97
2, 1, 0.23
2, 10, 4.62
2, 100, 19.26
2, 1000, 17.91
2, 10000, 34.34
2, 100000, 49.83
2, 500000, 49.27
4, 1, 0.34
4, 10, 0.259
4, 100, 16.7
4, 1000, 38.66
4, 10000, 82.39
4, 100000, 91.09
4, 500000, 91.49
8, 1, 0.26
8, 10, 2.39
8, 100, 16.21
8, 1000, 48.49
8, 10000, 137.59
8, 100000, 166.17
8, 500000, 181.62
flip3 155%
```

	A	B	C
1	1	1	1.44
2	1	10	3.99
3	1	100	8.07
4	1	1000	9.33
5	1	10000	23.4
6	1	100000	25.13
7	1	500000	25.97
8	2	1	0.23
9	2	10	4.62
10	2	100	19.26
11	2	1000	17.91
12	2	10000	34.34
13	2	100000	49.83
14	2	500000	49.27
15	4	1	0.34
16	4	10	0.259
17	4	100	16.7
18	4	1000	38.66
19	4	10000	82.39
20	4	100000	91.09
21	4	500000	91.49
22	8	1	0.26
23	8	10	2.39
24	8	100	16.21
25	8	1000	48.49
26	8	10000	137.59
27	8	100000	166.17
28	8	500000	181.62

Oregon State University
Computer Graphics

mjb – March 9, 2023

A Warning about Editing on Windows and Running on Linux

Some of you will end up having strange, unexplainable problems with your csh scripts or .cpp prograns. This could be because you are typing your code in on Windows (using Notepad or Wordpad or Word) and then running it on Linux. Windows likes to insert an extra carriage return ('\r') at the end of each line, which Linux interprets as a garbage character.

You can confirm this by typing the Linux command:
od -c loop.csh
which will show you all the characters, even the '\r' (carriage returns, which you don't want) and the '\n' (newlines, which you do want).

To get rid of the carriage returns, enter the Linux command:
tr -d '\r' < loop.csh > loop1.csh
Then run loop1.csh

This works too:
sed -i -e 's/\r$//' loop.csh

Or, on some systems, there is a utility called *dos2unix* which does this for you:
dos2unix < loop.csh > loop1.csh

Sorry about this. Unfortunately, this is a fact of life when you mix Windows and Linux.

Oregon State University
Computer Graphics

mjb – March 9, 2023